Inleiding tot Python

Brecht Baeten

28 september 2015

1 Wat is Python?

Python is een open source programmeer taal. Voor ingenieurstoepassingen of wetenschappelijke projecten is Python interessant aangezien er een aantal Python modules bestaan die hier specifiek op gericht zijn. Deze modules maken het beheren van data of uitvoeren van complexe wiskundige algoritmes zeer eenvoudig, vandaar de populariteit. Python werkt volledig object georiënteerd, dit laat toe efficiente en leesbare code te schrijven. Het schijven van leesbare code is één van de hoekstenen van Python aangezien code veel vaker gelezen wordt dan geschreven. Python code hoeft ook niet gecompileerd te worden (althans niet door de gebruiker). Python is een geïnterpreteerde programmeertaal wat inhoudt dat code rechtstreeks uitgevoerd kan worden zonder eerst te compileren. Een ander groot voordeel van Python boven bijvoorbeeld Matlab is dat het gratis is! Iedereen, wetenschappers, studenten, particulieren, grote of kleine bedrijven kan Python gratis downloaden en gebruiken. In veel Linux distributies word Python standaard meegeleverd wat zelfs de installatie overbodig maakt.

Een nadeel van Python ten opzichte van Matlab is dat de verschillende beschikbare IDE's (Integrated Development Environments) veel minder vloeiend en gebruiksvriendelijk zijn dan de bekende Matlab interface. Dit is echter slechts een klein nadeel ten opzichte van de vele voordelen van Python.

De broncode van deze tekst alsook alle voorbeelden kunnen worden gedownload via `https://github.com/BrechtBa/inleiding-tot-python/`.

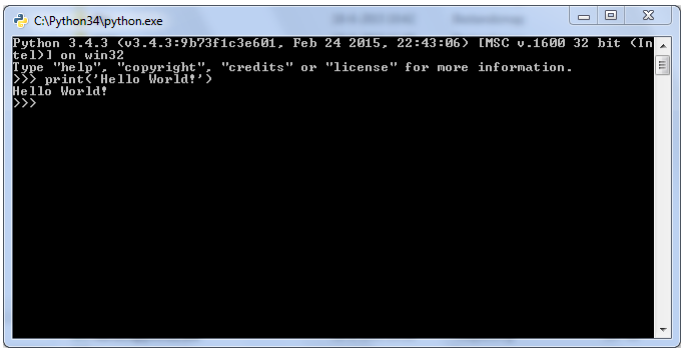
2 Installatie in Windows

Alle nog ondersteunde versies van Python kunnen worden gedownload via `https://www.python.org/downloads/`. Hier vind je de broncode en installer executables voor verschillende platformen. Downloaden, uitvoeren en klaar. Het is wel aan te randen om niet de nieuwste versie te installeren aangezien verschillende modules een tijdje nodig hebben om hun code aan te passen naar een nieuwe versie. Tijdens de installatie is het interessant om Python aan je Windows search path toe te voegen, zo kan je python steeds van in een command line interface openen. Het is ook interessant om de installatie locatie te veranderen naar bijvoorbeeld "C://python34".

Je kan nu al Python starten door op python.exe te klikken. Er opent een console waarin je python commando's kan invoeren (zie Figuur 1). Typ hier bijvoorbeeld:

```
print('Hello World!')
```

Druk op enter en je hebt je eerste python commando uitgevoerd.



Figuur 1: Python console met eerste commando

Python code kan opgeslagen worden in eenvoudige tekst bestanden met de extentie ".py". Om deze te schrijven is een goede text editor met syntax highlighting aan te raden. In Windows is *notepad++* een goed, open source alternatief met syntax highlighting voor Python en een groot aantal andere talen ingebouwd. *notepad++* kan worden gedownload via `https://notepad-plus-plus.org/download/`.

Omdat Python vanuit een command line interface zal worden aangeroepen is het interessant om een deftige console te installeren. *ConEmu* is een open source console emulator met gelijkenissen aan *bash* in Linux. Deze is te downloaden vanaf `http://sourceforge.net/projects/conemu/files/latest/download`. Simpelweg downloaden, unzippen in een folder naar keuze en klaar. De standaard *cmd* console kan echter ook gebruikt worden.

3 Scripts

Omdat het telkens opnieuw invoeren van commando's achter elkaar niet zo praktisch is, is het interessant om een reeks commando's op te slaan in een script en dit uit te voeren. Open een tekst-editor en typ "`print('Hello World!')`" en sla het bestand op als hello_world.py. Om het bestand uit te voeren open je een console, navigeer naar de locatie waar je het bestand hebt opgeslagen, typ "`python hello_world.py`" en druk op enter. Het resultaat is hetzelfde als daarnet.

Python heeft een aantal interessante ingebouwde variabele types. Open een nieuw bestand, typ onderstaande commando's en sla het op als variables.py.

```
# integer
A = 1
print(A)

# float
B = 3.572
print(B)

# string
C = 'string'
print(C)

# list
```

```
D = [1,2,3]
print(D)
E = [1,'test',4,D]
print(E)
print(E[:2])
print(E[-1])

# dictionary
F = {3:1, 5:'test', 'spam':'eggs'}
print(F)
print(F['spam'])

# assignment by reference
G = F
G['spam'] = 'african swallow'
print(F)
print(G)

H = dict(F)
H['spam'] = 'european swallow'
print(F)
print(H)
```

Voer het script ditmaal uit met het commando "`python -i variables.py`". De "`-i`" zorgt ervoor dat python na het uitvoeren van alle commando's in het script naar de interactieve console gaat. Alle reeds gedefinieerde variabelen zijn nu ook beschikbaar in de console. Typ bijvoorbeeld:

```
E[3]
```

of:

```
A+B
```

Door "`Ctrl+Z`" in te geven kan je Python afsluiten en terugkeren naar de command line interface.

Zoals te merken in het voorgaande voorbeeld is Python zeer flexibel met data types. Zo kunnen een *Integer* en een *Float* met elkaar opgeteld worden en kan een *List* elementen met verschillende datatypes bevatten. Een zeer interessant data type is het *Dictionary* of *dict*. Hierin kunnen key / value paren worden opgeslagen en terug opgeroepen. Zowat elke Python variabele kan een *Dictionary* key zijn wat dit type zeer flexibel maakt. Het is gewenst om even in te gaan op de manier van indexeren in Python. Een Python *List* start bij index 0. Het laatste element kan je oproepen met de index -1, het voorlaatste met index -2, enz. Je kan een deel van de lijst opvragen met behulp van de "`:`" operator, de subset loopt dan van de index voor de "`:`" tot (niet tot en met) de index erna. Indien er geen index voor of achter de "`:`" staat zal de subset starten of eindigen bij respectievelijk het eerste of het laatste element.

Er moet even aandacht besteed worden aan de manier waarop variabelen in Python gebruikt worden. In Python zijn variabelen slechts namen die naar een bepaalde waarde verwijzen. Wanneer een variabele toegewezen wordt, wordt enkel de referentie naar de waarde toegewezen. In het voorbeeld hierboven zullen "`print(F)`" en "`print(G)`" hetzelfde resultaat geven. Hierboven wordt "`H`" echter gedefinieerd als een nieuwe *Dictionary* met dezelfde waarden als "`F`". De commando's "`print(F)`" en "`print(H)`" zullen dus een verschillend resultaat hebben.

Echt programmeren begint pas bij maken van lussen en voorwaarden: flow control. In python is dit zeer eenvoudig met behulp van "`for`" lussen of "`if else`" structuren:

```
for i in range(10):
    if i%2 == 0:
        print('{} is even'.format(i))
    else:
        print('{} is oneven'.format(i))
```

In Python moet elk flow control element eindigen met een ":". De code binnen het element moet een tab inspringen. Dit zorgt voor een zekere leesbaarheid in de code. De "`range(10)`" functie in de "**`for`**" lus maakt een *List* met waarden van 0 tot en met 9. De "`.format(i)`" functie formatteert zijn inhoud op de plaats van de accolade's in de voorafgaande string. De format functie is eigenlijk een methode van het *String* datatype en geeft een hele reeks formatteer opties. Even zoeken op het net leert je heel veel over zulke ingebouwde functies.

Je kan over zowat elk datatype itereren. Een lus over een dictionary kan bijvoorbeeld op verschillende manieren. Ook itereren over verschillende variablen tegelijk is eenvoudig met de "`zip()`" functie. Indien je itereert over een *List* en de index van de variabelen wil gebruiken kan je de functie "`enumerate()`" gebruiken:

```
A = {'foo': 'bar',
     'spam': 'eggs',
     'bacon': 'spam'}

# dictionaries
for key in A:
    print('{}: {}'.format(key,A[key]))

for key,val in A.iteritems():
    print('{}: {}'.format(key,val))

# zip
B = [1,2,3]
C = [7,8,9]
for b,c in zip(B,C):
    print('b = {:.0f}, c = {:.3f}'.format(b,c))

# enumerate
D = [1,11,111,1111]
for i,d in enumerate(D):
    print('D[{}] = {}'.format(i,d))
```

Een set commando's die vaak op dezelfde manier gebruikt worden kan je groeperen in een functie. Een functie definieer je met het "**`def`**" keyword, gevolgd door de functie naam, de argument namen tussen haakjes en een dubbelpunt. Argumenten kunnen verplicht of optioneel zijn. voor optionele argumenten moet een default waarde opgegeven worden in de functie definitie. Alle verplichte argumenten moeten ook voor de optionle komen in de functie definitie. Wanneer een functie veel optionele argumenten heeft is het gemakkellijk om deze via naam=waarde paren op te geven, de volgorde is dan niet meer van belang:

```
def spam(A,B,C=0,D=0):
    val = 100*A+10*B+C+0.1*D
    return val

D = spam(4,1)
print(D)

E = spam(4,1,5.3)
```

```
print(E)

F = spam(4,1,D=3,C=5)
print(F)
print(val)
```

Variabelen die binnen een functie gedefinieerd zijn bestaan ook enkel in de functie namespace. Bovenstaande code geeft dus een error omdat `val` enkel in de functie namespace gedefinieerd is, niet in de globale namespace. Functies kunnen wel gebruik maken van variabelen in de globale namespace maar niet omgekeerd. Wanneer de bovenstaande code runt krijg je dan ook de volgende foutmelding:

```
Traceback (most recent call last):
  File "C:\examples\functions.py", line 10, in <module>
    print(val)
NameError: name 'val' is not defined
```

4 NumPy, SciPy en Matplotlib

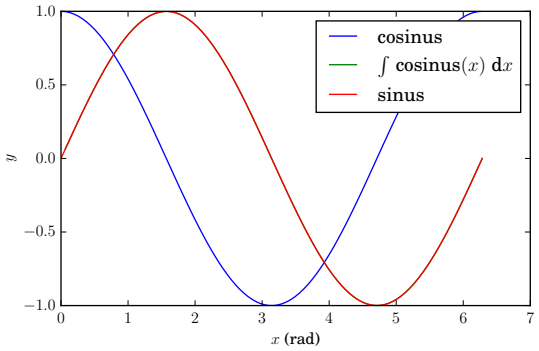
Er bestaan enorm veel modules waarin specifieke functie gedefinieerd zijn voor gebruik in Python. Drie voor Ingenieurs interessante modules zijn *NumPy*, een lineair algebra module, *SciPy*, een algemeen wetenschappelijke wiskunde module en *Matplotlib*, een module voor het maken van figuren van hoge kwaliteit. Windows installers kunnen worden gedownload via `http://sourceforge.net/projects/numpy/files/NumPy/`, `http://sourceforge.net/projects/scipy/files/scipy/` en `http://matplotlib.org/downloads.html`. Kies de binary die overeenkomt met jou systeem architectuur en python versie, downloaden en installeren. De meeste modules lopen steeds één versie achter op de laatste Python release, het kan dus interessant zijn om een iets oudere versie van Python te installeren.

Om een module te kunnen gebruiken in een script moet deze eerst geïmporteerd worden. Dit gebeurt met een "**`import`**" statement en kan op verschillende manieren. In Python blijft elke geïmporteerde module zijn eigen namespace behouden. Je kan dus de `linspace` funtie uit *NumPy* aanroepen als "`numpy.linspace(...)`". Om de notatie wat te verkorten kan je een module importeren onder een zelf gekozen naam met "**`import`** `...` `as` `...`" zoals aangegeven in het voorbeeld hieronder. Voor meer info over *NumPy* of *SciPy* wordt verwezen naar de online documentatie op `http://docs.scipy.org/doc/numpy/` en `http://docs.scipy.org/doc/scipy/reference/`

Matplotlib kan LaTeX gebruiken om tekst weer te geven, dit kan permanent gedaan worden door de 'rc' parameters aan te passen in het matplotlibrc bestand of deze in het script zelf aan te passen zoals hieronder aangegeven. De grootte van de figuur wordt in inch opgegeven vandaar een correctiefactor naar centimeter. Met behulp van de "`savefig`" functie wordt de figuur opgeslagen als pdf (Figuur 2). Een zeer groot aantal voorbeelden van *Matplotlib* figuren en de code gebruikt om ze te maken is terug te vinden op `http://matplotlib.org/gallery.html`.

```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt

x = np.linspace(0,2*np.pi,100)
c = np.cos(x)
s = np.sin(x)
```

```
ss = scipy.integrate.cumtrapz(c,x)

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('figure', autolayout=True)

plt.figure(figsize=(15/2.54,10/2.54))
plt.plot(x,c,label=r'cosinus')
plt.plot(x[1:],ss,label=r'$\int$ cosinus$(x)$ d$x$')
plt.plot(x,s,label=r'sinus')
plt.gca().set_xlabel(r'$x$ (rad)')
plt.gca().set_ylabel(r'$y$')
plt.legend()

plt.savefig('sinus_cosinus.pdf')
plt.show()
```



Figuur 2: Matplotlib voorbeeld

5 Object georiënteerd

Een grote sterkte van Python is de zeer grote object gerichtheid van de programeertaal. Zowat alles in Python is een object met zijn methodes en kan aangepast of gebruikt worden in child klasses. Een klasse wordt in python gedefinieerd met het keyword "**`class`**" gevolgd door de klassenaam het basis object tussen haakjes en een dubbelpunt. Opnieuw dient alles wat binnen de klasse definitie valt één tab verschoven te zijn tenopzichte van de klasse definitie. Binnen een klasse zijn er een aantal belangrijke methodes die kunnen gebruikt worden. De eerste is de "`__init__`" methode. Deze wordt gebruikt om een object instance te creëren. Het eerste argument is steeds "`self`", dit wordt gebruikt om attributen aan de instance toe te kennen, verder kunnen de argumenten op dezelfde wijze als bij functies gedefineerd worden. De "__" voor en na de methode definitie geven aan dat deze een ingebouwde functie heeft. In het voorbeeld hieronder wordt ook de "`__str__`" methode gedefinieerd die het printen van het object definieert. Telkens is "`self`" het eerste argument om een referentie naar de huidige object instance te hebben.

In het voorbeeld hieronder wordt ook een `density` methode gedefinieerd die gebruikt kan worden om de dichtheid van het gas te berekenen. Onmiddellijk na de definitie volgt de documentatie van de functie tussen drie aanhalingstekens "`"""`". Deze wordt weergegeven indien de de "`help()`" functie wordt opgeroepen, dit kan trouwens ook bij gewone functies. Het is een goede gewoonte om

hier steeds te schijven wat de functie doet, wat ze returnt, wat de argumenten betekenen en welk datatype de argumenten moeten zijn. Een andere goede gewoonte is om methodes die bedoeld zijn om enkel binnen de klasse definitie te gebruiken te beginnen met een "_". Op die manier is het voor een gebruiker van de klasse meteen duidelijk dat deze methode niet voor hem of haar bedoeld is.

In de code hieronder is het gedeelte dat rechtstreeks uitgevoerd moet worden in een "`if __name__ == '__main__':`" geplaatst. Dit zorgt ervoor dat deze code enkel uitgevoerd wordt als het script rechtstreeks wordt uitgevoerd. Meer hierover in de volgende sectie.

```
class IdealGas(object):
    """
    A class defining an ideal gas
    """
    def __init__(self,name,R):
        """
        Returns a Gas Instance

        Arguments:
        name: string, name of the gas
        R: float, specific gas constant
        """
        self.name = name
        self.R = R

    def __str__(self):
        return '{}, R={} J/kgK'.format(self.name,self.R)

    def density(self,P,T):
        """
        Returns the density of the gas in kg/m3

        Arguments:
        P: float, pressure in Pa
        T: float, Temperature in K
        """
        return P/self.R/T

if __name__ == '__main__':
    help(IdealGas.density)

    a = IdealGas('air',287)
    print(a)
    print('rho = {:.3f} kg/m3'.format(a.density(101325,293.15)))
```

Een belangrijk element van object georiënteerd programeren is inheritance. In Python is dit opniew zeer eenvoudig. In het voorbeeld hieronder wordt een "PerfectGas" klasse gedefinieerd op basis van van de eerder gemaakte "IdealGas" klasse. De eerste regel zorgt ervoor dat de "IdealGas" klasse beschikbaar is in dit script, meer hierover in Sectie 6. We kunnen nu methodes van de parent klasse herdefiniëren of nieuwe methodes toevoegen. In het voorbeeld hieronder wordt de "`__str__`" methode aangepast door iets toe te voegen aan de originele methode met behulp van de "`super`" functie. Merk op dat alle methodes van de parent klasse nog steeds beschikbaar zijn.

```
from object_oriented import IdealGas

class PerfectGas(IdealGas):
    """
    A class defining a perfect gas as an ideal gas with constant cp
```

```
    """
    def __init__(self,name,R,cp):
        """
        Returns a PerfectGas Instance

        Arguments:
        R: float, specific gas constant
        cp: float, heat capacity at constant pressure
        """
        IdealGas.__init__(self,name,R)
        self.cp = cp
        self.cv = self.cp-self.R

    def __str__(self):
        return super(PerfectGas,self).__str__() + ', cp={} J/kgK'.format(self.cp)

    def du(self,dT):
        """
        Returns the change in specific energy of the gas in J/kg

        Arguments:
        dT: float, temperature difference in K
        """
        return self.cv*dT


if __name__ == '__main__':

    a = IdealGas('air',287)
    b = PerfectGas('air',287,1004)

    print(a)
    print(b)

    print('rho = {:.3f} kg/m3'.format(a.density(101325,293.15)))
    print('rho = {:.3f} kg/m3'.format(b.density(101325,293.15)))

    print('du = {:.2f} J/kg'.format(b.du(50)))
```

De bedoeling van inheritance is om zoveel mogelijk code te hergebruiken, en dus geen code dubbel te schrijven, wat moeilijk te onderhouden is.

6 Eigen modules schrijven

Wanneer een project groeit komt er een moment dat het niet meer interessant is om alle code in één file the schrijven. In Python kan dit door het project op te delen in modules en deze in een hoofd bestand te importeren en te gebruiken. Een Python module kan je definiëren door in een folder een bestand "__init__.py" aan te maken. In dit bestand kan je code zetten die uitgevoerd dient te worden tijdens de initialisatie van de module, maar je kan deze ook gewoon leeg laten. Wanneer er een "__init__.py" in een folder staat kan je de verschillende files in die folder importeren via het "**import**" commando zoals in het voorbeel hierboven en hieronder.

Wanneer je dit doet zullen alle commando's uit het geïmporteerde script uitgevoerd worden en de functie of klasse definities uit die file worden beschikbaar via de " ." notatie. Nu wordt ook duide-

lijk waarom hierboven een deel van de code binnen een "`if __name__ == '__main__':`" constructie geplaatst is. Wanneer een script rechtstreeks uitgevoerd wordt zal Python een "`__name__`" veranderlijke aanmaken met de waarde "`'__main__'`". Wanneer een script geïmporteerd wordt, wordt in de namespace van die module de veranderlijke "`__name__`" aangemaakt met de naam van de file als waarde. Dit kan interessant zijn bij het testen van submodules zoals hierboven. Je kan steeds kijken welke variabelen er gedefinieerd zijn in de huidige namespace met de "`dir()`" functie.

In het voorbeeld hieronder wordt ook een submodule uit de subfolder mymodule aangeroepen. Deze bevat een functie die opnieuw via de "." notatie kan worden gebruikt. Via de "`dir(mymodule.submodule)`" functie kan je ook achterhalen welke functies allemaal gedefinieerd zijn in een bepaalde module. Wanneer een folder geïmporteerd wordt worden enkel de commando's in het "__init__.py" script in die folder uitgevoerd. Je kan dan bijvoorbeeld in dit script een aantal andere scripts importeren die zo ook beschikbaar worden in het hoofd script.

```
import object_oriented
import mymodule.submodule

# what is in the current namespace
print( dir() )
print( dir(mymodule.submodule) )

# a file imported from the current directory
a = object_oriented.IdealGas('air',287)
b = object_oriented.IdealGas('argon',208)
print(a.density(101325,293.15))
print(b.density(101325,293.15))

# a file imported from a subdirectory
mymodule.submodule.test()
```

Python bied ook de mogelijkheid om alle definities uit een bepaalde module te importeren in de huidige namespace. Dit gebeurt met het "`from mymodule.submodule import *`" commando. Eens dit is uitgevoerd zijn alle functies klasses en variabelen gedefinieerd in "mymodule\submodule.py" rechttreeks beschikbaar, zonder "." notatie.

Door op deze manier een project goed te structureren blijft het geheel leesbaar, overzichtelijk en goed te onderhouden.